CMPS 312



Data Management



Dr. Abdelkarim Erradi
CSE@QU

Outline

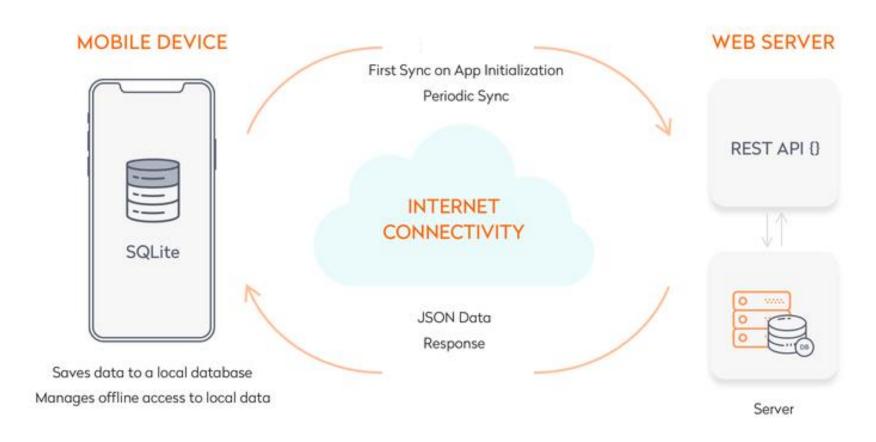
- Data persistence options on Android
- 2. Room programming model
- 3. Type Converters & Foreign keys
- 4. Observable Queries using LiveData / Flow

Data persistence options on Android





Offline app with Sync



 Cache relevant pieces of data on the device. App continues to work offline when a network connection is not available.



 When the network connection is back, the app's repository syncs the data with the server.

Data Storage Options on Android

Preferences DataStore

- Lightweight mechanism to store and retrieve key-value pairs
- Typically used to store application settings (e.g., app theme, language), store user details after login

Files

 Store unstructured data such as text, photos or videos, on the device (Current application folder only) or removable storage

• SQLite database

Store structured data (e.g., posts, events) in tables

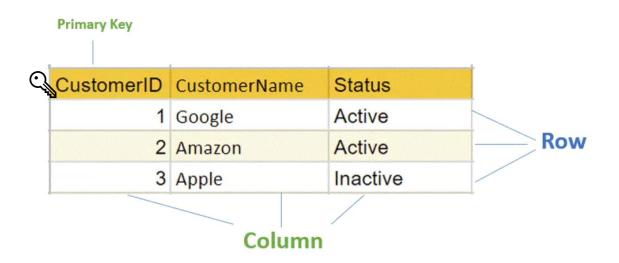
Cloud Data Stores

o e.g.,



Relational Database

- Database allows persisting structured data
- A relational database organizes data into tables
 - A table has rows and columns
 - Tables can have relationships between them
- Tables could be queries and altered using SQL



SQL Statements

- Structured Query Language (SQL)
 - Language used to define, query and alter database tables
 - SQL is a language for interacting with a relational database
- Creating data:

```
INSERT into Person (firstName, lastName)
VALUES ("Ahmed", "Sayed")
```

Reading data:

```
SELECT * FROM Person WHERE lastName = "Sayed"
```

Updating data:

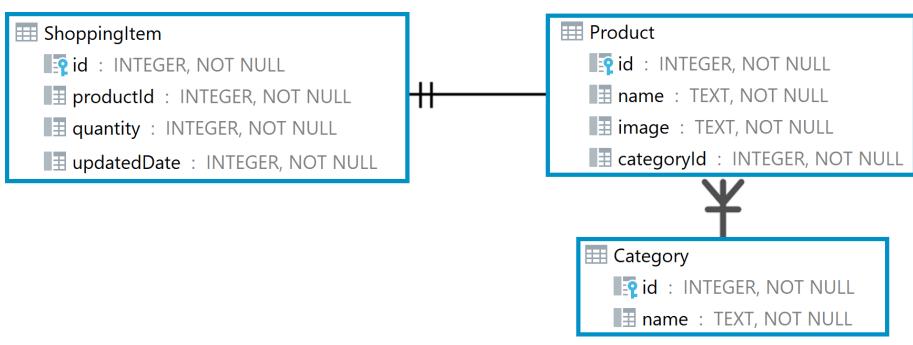
```
UPDATE Person SET firstName = "Ali" where
   lastName = "Sayed"
```

Deleting data:

```
DELETE from Person where lastName = "Sayed"
```

Database Schema of Shopping List App

- The Entity Relationship (<u>ER</u>) diagram of the Shopping List App database
 - A ShoppingItem has an associated Product
 - Product has a Category
 - Category has many products



Querying Multiple Tables with Joins

- Combine rows from multiple tables by matching common columns
 - For example, return rows that combine data from the Product and Category tables by matching the Product.categoryId foreign key to the Category.id primary key

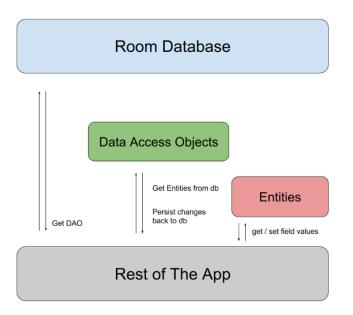
select p.id, p.name, p.image, c.name category

from Product p **join Category c on** p.categoryId = c.id

where p.categoryld = '1'

id					
2 Melon Fruits 3 Watermelon Fruits 4 Banana Fruits 5 Pineapple Fruits 6 Mango Fruits 7 Red Apple Fruits 8 Green Apple Fruits 9 Pear Fruits	id	\$	name	image	category
3 Watermelon ♥ Fruits 4 Banana	1		Grapes	Ý	Fruits
4 Banana Fruits 5 Pineapple Fruits 6 Mango Fruits 7 Red Apple Fruits 8 Green Apple Fruits 9 Pear Fruits	2		Melon	•	Fruits
5 Pineapple & Fruits 6 Mango	3		Watermelon	w	Fruits
6 Mango	4		Banana	2	Fruits •
7 Red Apple Fruits 8 Green Apple Fruits 9 Pear Fruits	5		Pineapple	š	Fruits
8 Green Apple 🖒 Fruits 9 Pear Fruits	6		Mango		Fruits
9 Pear Fruits	7		Red Apple	\tilde{\	Fruits
	8		Green Apple	$\overset{\bullet}{\Box}$	Fruits
10 Peach 🍎 Fruits	9		Pear	•	Fruits
	10	, ma	Peach	Ď	Fruits

Room programming model





Room Library

- The Room persistence library provides an abstraction layer over SQLite to ease data management
 - Define the database, its tables and data operations using annotations
 - Room automatically translates these annotations into SQLite instructions/queries to be executed by the DB engine

Dependencies:

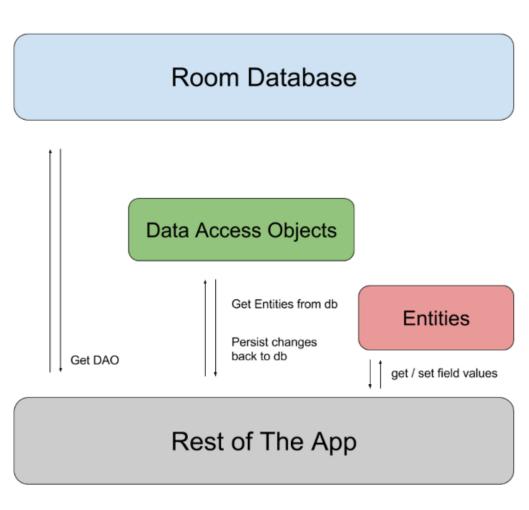
```
def room_version = "2.3.0"
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
// Kotlin Extensions and Coroutines support for Room
implementation "androidx.room:room-ktx:$room_version"
```

Room architecture diagram

Working with Room

- Model DB Tables as regular entity classes
- Define queries for Select, Insert, Update and Delete and associate them with methods in **DAO interface**. Implementation is autogenerated by the complier
- Interact with the database using DAOs
- RoomDatabase → holds a connection to the SQLite DB and all the operations are executed through it

3 major components in Room



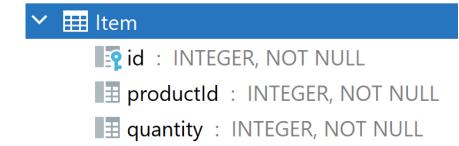
Room main components

- - Kotlin class annotated with @Entity to map it to a DB table
 - Must specify one of the entity properties as a primary key
 - Table representation (e.g., name and column names) is controlled by annotations
- Data Access Object (DAO) → how to retrieve and persist entities
 - Contains CRUD methods defining operations to be done on data
 - Interface or abstract class marked as @Dao
 - One or many DAOs per database
- Database → where data is persisted
 - abstract class that extends RoomDatabase and annotated with @Database

Entity

- Entity represents a database table, and each entity instance corresponds to a row in that table
 - Class properties are mapped to table columns
 - Each entity object has a Primary Key that Uniquely identifies the entity object in memory and in the DB
 - The primary key values can be assigned by the database by specifying autoGenerate = true

```
@Entity
data class Item(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0
    val productId: Long,
    var quantity: Int)
```



Customizing Entity Annotations



- By default, the name of the entity class is the same as the associated table and the name of table columns are the same as the class properties
 - In most cases, the defaults are sufficient but can be customized
 - Use @Entity (tableName = "...") to set the name of the table
 - The columns can be customized using @ColumnInfo(name = "column_name") annotation
- If an entity has properties that you don't want to persist, you can annotate them using @lgnore

DAO @Query

- @Query used to annotate query methods
- Room ensures compile time verification of SQL queries

```
@Dao
interface UserDao {
    @Query("select * from User limit 1")
    suspend fun getFirstUser(): User
    @Query("select * from User")
    suspend fun getAll(): List<User>
    @Query("select firstName from User")
    suspend fun getFirstNames(): List<String>
    @Query("select * from User where firstName = :fn")
    suspend fun getUsers(fn: String): List<User>
    @Query("delete from User where lastName = :ln")
    suspend fun deleteUsers(ln: String): Int
```

DAO @Insert, @Update, @Delete

- Used to annotate insert, update and delete methods
- Suspend ensure that DB operations are not done on the main UI thread

```
@Dao
interface UserDao {
    @Insert
    suspend fun insert(user: User): Long
    @Insert
    suspend fun insertList(users: List<User>): List<Long>
    @Delete
    suspend fun delete(user: User)
    @Delete
    suspend fun deleteList(users: List<User>)
    @Update
    suspend fun update(user: User)
    @Update
    suspend fun updateList(users: List<User>)
```

Room database class

- Abstract class that extends RoomDatabase and Annotated with @Database
- Provides a singleton dbInstance object created using Room.databaseBuilder()
 to create (if does not exit) and connect to the database
- Serves as the main access point to get DAOs to interact with DB

```
@Database(entities = [Item::class], version = 1)
abstract class ShoppingDB : RoomDatabase() {
    companion object { // Create a singleton dbInstance
        private var dbInstance: ShoppingDB? = null
        fun getInstance(context: Context): ShoppingDB {
            if (dbInstance == null) {
                dbInstance = Room.databaseBuilder(
                    context,
                    ShoppingDB::class.java, "shopping.db"
                ).fallbackToDestructiveMigration().build()
            return dbInstance as ShoppingDB
```

Type Converters & Foreign keys





TypeConverter

- SQLite only support basic data types, no support for data types such as Date, enum, BigDecimal etc. Need to add a TypeConverter for such data types
- Convert an entity property datatype to a type that can be written to the associated table column and vice versa

```
class DateConverter {
    // Convert from Date to a value that can be stored in SQLite Database
    @TypeConverter
    fun toLong(date: Date) : Long = date.time

    // Convert from date Long value read from SQLite DB to a Date value
    // that can be assigned to an entity property
    @TypeConverter
    fun toDate(dateLong: Long) : Date = Date(dateLong)
}

@TypeConverters(DateConverter::class)
abstract class ShoppingDB : RoomDatabase() { ... }
```

Nested @Embedded object

- Nested object annotated with @Embedded -> Room flattens out the embedded object and maps its properties to columns in the DB table
 - User table will have a houseNumber, street and city columns

```
data class Address(val houseNumber: String,
                      val street: String,
                      val city: String)
@Entity
                                              User
data class User (
    @PrimaryKey(autoGenerate = true)
                                                id: INTEGER, NOT NULL
                                                firstName: TEXT, NOT NULL
    var id: Long,
                                                lastName: TEXT, NOT NULL
    val firstName: String,
                                                houseNumber: TEXT, NOT NULL
    val lastName: String,
                                                street: TEXT, NOT NULL
    @Embedded val address: Address
                                                city: TEXT, NOT NULL
```

Enforce integrity checks with foreign keys

- Foreign key allows integrity checks (e.g., can insert pet only for a valid owner) & cascading deletes
 - onDelete = ForeignKey.CASCADE when owner is deleted then auto-delete associated pets

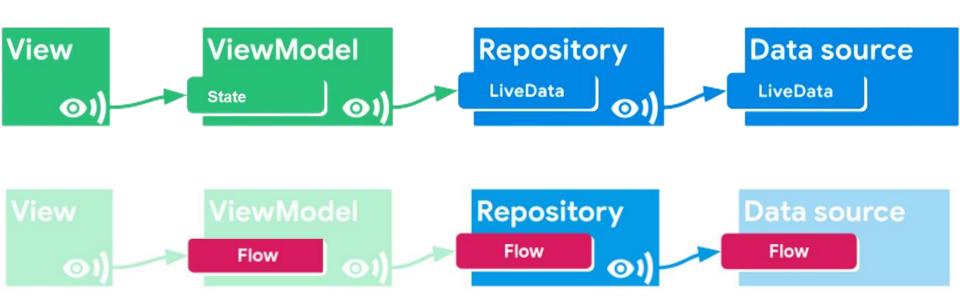
```
@Entity(foreignKeys = [
        ForeignKey(entity = Owner::class,
                parentColumns = ["id"],
                childColumns = ["ownerId"],
                onDelete = ForeignKey.CASCADE)
        ],
    // Create an index on the ownerId column to speed-up query execution
    indices = [Index(value = ["ownerId"])])
data class Pet(@PrimaryKey val id: Long,
               val name: String, val ownerId: Long)
@Entity
data class Owner(@PrimaryKey val id: Long, val name: String)
```

Simplify one-to-many queries via @Relation

Use @Relation annotation to eagerly fetch associated relations

```
data class OwnerWithPets(
       @Embedded
        val owner: Owner,
       @Relation( parentColumn = "id", entityColumn = "ownerId" )
        val pets: List<Pet>
 @Dao
 public interface OwnerDao {
     // Entity and its relations are fetched by Room
     @Transaction
     @Query("SELECT * FROM Owner")
     suspend fun getOwnerWithPets() : List<OwnerWithPets>
```

Observable Queries using LiveData / Flow







Observable Queries

- Observable queries allow automatic notifications when data changes
 - Notifies the app with of any data updates
 - Can implemented using LiveData or Flow
- We can accomplish this using LiveData, a lifecycleaware observable value holder
 - We simply wrap the return type of our DAO methods with LiveData

```
// App will be notified of any changes of the Item table data
// Whenever Room detects Item table data change our LiveData
observer will be called with the new list of items
// No need for suspend function as LiveData is already asynchronous
@Query("Select * from Item")
fun getAll() : LiveData<List<Item>>
```

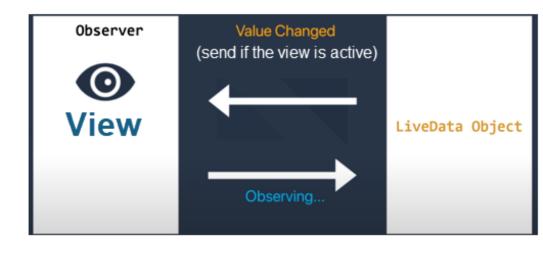
LiveData

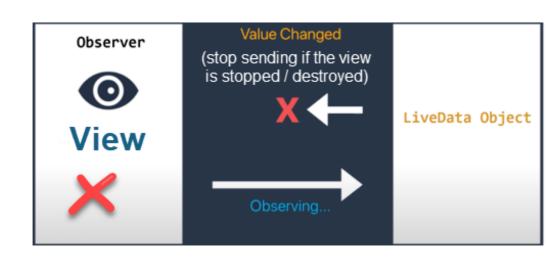
- <u>LiveData</u> is an <u>observable data holder</u> (typically returned by the Repository): <u>active</u> observers (i.e., the ViewModel) get notified when data change
- The Repository return LiveData objects that the ViewModel can observe and notify the Ul accordingly
 - This decouples the Repository from the ViewModel: The Repository does NOT have any direct reference to the ViewModel

LiveData is lifecycle-aware

LiveData is aware of the Lifecycle of its Observer

- Notifies data changes to only active observers (Stopped/Destroyed View will NOT receive updates)
- It automatically removes the subscription when the observer is destroyed so it will not get any updates

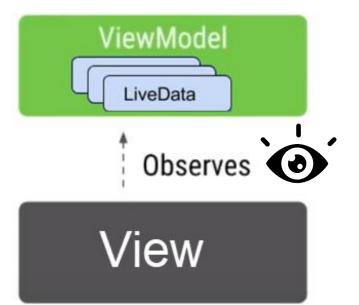




LiveData in Code

LiveData warps around an object and allows the view to observe it





 ViewModel expose LiveData objects that the View can observe

```
object DataRepository {
    // In a real app the LiveData will be returned by
    // a database or Web API as we will see later
    fun getRedCardsCount(): LiveData<Int> = liveData {
        emit(2)
    }
}
```

```
class ScoreViewModel : ViewModel() {
   val redCardsCount = DataRepository.getRedCardsCount()
}
```

View observes LiveData changes

LiveData.observeAsState()

- LiveData.observeAsState() registers as a listener and represent the values as a State
 - Whenever a new value is emitted, Compose recomposes those parts of the UI where that state.value is used
 - Required dependency in build.gradle:

implementation "androidx.compose.runtime:runtime-livedata:\$compose_version"

What is Flow?









- Stream of values (produced one by one over time instead of all at once)
 - as opposed to functions that return only a single value
 - Values could be generated from network requests or database calls
- Can transform a flow using operators like map, filter, etc.

```
fun stream(): Flow<String> = flow {
    emit("@") // Emits the value upstream @
    emit("@")
    emit(">")
}
```



```
object WeatherRepository {
    private val weatherConditions = listOf("Sunny", "Windy", "Rainy", "Snowy")
    fun getWeather(): Flow<String> =
        flow {
        var counter = 0
        while (true) {
            counter++
            delay(3000)
            emit(weatherConditions[counter % weatherConditions.size])
        }
}
```

```
class WeatherViewModel : ViewModel() {
   val weatherFlow: Flow<String> = WeatherRepository.getWeather()
}
```

```
@Composable
fun ScoreScreen() {
    val weatherViewModel = viewModel<WeatherViewModel>()

    val weatherFlow = weatherViewModel.weatherFlow.CollectAsState(initial = "")

    // Recomposes whenever redCardsCount changes

    Text(
        text = "Red cards count: ${weatherFlow.value}" )
    } ... }
```

Flow Operators

 Flow has operators similar to collections such as map, filter and reduce

```
(1..5).asFlow()
    .filter { it % 2 == 0 }
    .map { it * it }
    .collect { println(it.toString()) }
val result =(1...5).asFlow()
                   .reduce { a, b -> a + b }
println("result: $result")
```

Summary

Major Components

- @Entity Defines table structure
- @Dao An interface with functions define how to read/write from the database
- @Database Serves as the main access point to get DAOs to interact with DB



Resources

- Save data in a local database using Room
 - https://developer.android.com/training/data-storage/room

Room pro tips

 https://medium.com/androiddevelopers/7-pro-tips-forroom-fbadea4bfbd1

Room codelab

- https://codelabs.developers.google.com/codelabs/androidroom-with-a-view-kotlin/
- https://developer.android.com/codelabs/kotlin-androidtraining-room-database
- https://codelabs.developers.google.com/codelabs/androidpreferences-datastore